

Notes from Well House Consultants

These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit

*<http://www.wellho.net/net/whcotnl.html>
for details.*

1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

graham@wellho.net

technical contact

lisa@wellho.net

administration contact

Our full postal address is

404 The Spa
Melksham
Wiltshire
UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

Loops and Conditional Statements

Unless otherwise specified, code runs from top to bottom, very much in the manner that you read a book. Conditional statements such as "if" allow the programmer to choose whether or not blocks of the code are to be skipped over, and loops such as "while" allow blocks of code to be repeated.

<i>Booleans</i>	4
<i>"if" statement</i>	5
<i>"while" loop</i>	7
<i>"for" loop</i>	10
<i>Labels and breaks</i>	11

2.1 Booleans

We've read data, calculated with it and printed out the answers so far. A glorified calculator that performs exactly the same mechanical steps each time we run it.

But we are going to want to make decisions.

"If the temperature is high, print an advert for ice cream; otherwise, advertise thermal vests!!"

Decisions are based on something being "true" or "false" – that *something* is called a **boolean** object.

Boolean variables can be declared in just the same way as a **float** or an **int** or the others you have seen so far.

Boolean variables can have values assigned to them in just the same way that other variables can.

Beware: You can't assign a number to a boolean.

Fortunately, more operators are available which operate on numbers, but produce a boolean answer.

```
public class Updown
{
    public static void main(String[] args)
    {

        System.out.print("Yesterday: ");
        float yesterday=WellHouseInput.readNumber();
        System.out.print("Today: ");
        float today=WellHouseInput.readNumber();
        float av_temp;

        System.out.print("Av. temp. ");
        System.out.print(av_temp=(float)(
            (yesterday+today)/2.0));
        System.out.println(" degrees C");

        boolean chilling = (yesterday > today);
        System.out.print("\n\"It is getting colder\" is ");
        System.out.println(chilling);

    }
}
```

```
bash-2.04$ java Updown
Yesterday: 40
Today: 30
Av. temp. 35.0 degrees C
"It is getting colder" is true
bash-2.04$ java Updown
Yesterday: 35
Today: 48
Av. temp. 41.5 degrees C
"It is getting colder" is false
bash-2.04$
```

Figure 1 Running public class Updown

Booleans really do print out the word "true" or "false". There is no other value possible.

```
boolean chilling = (yesterday > today);
System.out.print("\nIt is getting colder\n is ");
System.out.println(chilling);
```

The operator `>` means "is greater than" and yields a boolean result. Other operators include:

```
<      less than
<=     less than or equal to
>=     greater than or equal to
==     equals
!=     does not equal
```

As you can see, some operators are now more than just a single character. In particular, you need to understand the difference between `=` and `==`.

```
=      evaluates the expression to the right of the sign and assigns the result to
       the variable named to the left.
==     evaluates two expressions, and if they yield the same result, it gives
       "true". If they do not yield the same result, it gives "false".
```

```
boolean temp_level = (yesterday == today);
```

- Compares the contents of the variables `yesterday` and `today`.
- Sets `temp_level` to "true" if they are the same [number], and to "false" otherwise.

By the way, did you notice?

```
System.out.print("\nIt is getting colder\n is ");
```

We wanted to include a double quote within the string. But a double quote is a special character and would terminate the string normally, so we use a backslash first.

What would `\\` print out?

`\n` prints a new line character.

2.2 "if" statement

We won't (very often!) want to just print out the words "true" or "false". Much more likely we'll want to perform an action *if* a boolean is true. Yes, an **if** statement.

```
public class Firstif
{
    public static void main(String[] args)
    {

        System.out.println("Let us warn you if it's getting colder!");
        System.out.print("Yesterday: ");
        float yesterday=WellHouseInput.readNumber();
        System.out.print("Today: ");
        float today=WellHouseInput.readNumber();

        boolean chilling = (yesterday > today);
        if (chilling)
            System.out.println("It is getting colder");
        System.out.println("Check completed");
    }
}
```

Figure 2 Running public class Firstif

```

bash-2.04$ java Firstif
Let us warn you if it's getting colder!
Yesterday: 40
Today: 30
It is getting colder
Check completed
bash-2.04$ java Firstif
Let us warn you if it's getting colder!
Yesterday: 30
Today: 40
Check completed
bash-2.04$

```

This is an **if** statement with a boolean value in round brackets after it.

```

if (chilling)
    System.out.println("It is getting colder");

```

- If that boolean value is true, then the following statement up to the **;** is performed.
- If the boolean value is false, that statement is skipped.

Once the **if** statement has been completed, execution proceeds with the next statement in the program in the usual way.

It can also be more flexible than that.

- You can put an expression that yields a boolean result in the brackets
- You can follow the **if** statement with a block¹ if you want more than one statement to be conditional
- You can provide an **else** block if you want to specify an "otherwise" set of statements

```

public class Secondif
{
    public static void main(String[] args)
    {
        System.out.println("Let us warn you if it's getting colder!");
        System.out.print("Yesterday: ");
        float yesterday=WellHouseInput.readNumber();
        System.out.print("Today: ");
        float today=WellHouseInput.readNumber();
        if (yesterday > today)
        {
            System.out.println("It is getting colder");
        }
        else
        {
            System.out.println("It is NOT getting colder");
            if (yesterday != today)
            {
                System.out.println("It is getting WARMER");
            }
            else
            {
                System.out.println("Static temperature");
            }
        }
    }
}

```

¹ a number of statements enclosed in braces

```

    }
    System.out.println("Check completed");
}
}

```

Figure 3 Running public class `Secondif`

```

bash-2.04$ java Secondif
Let us warn you if it's getting colder!
Yesterday: 30
Today: 33
It is NOT getting colder
It is getting WARMER
Check completed
bash-2.04$ java Secondif
Let us warn you if it's getting colder!
Yesterday: 33
Today: 30
It is getting colder
Check completed
bash-2.04$ java Secondif
Let us warn you if it's getting colder!
Yesterday: 33
Today: 33
It is NOT getting colder
Static temperature
Check completed
bash-2.04$

```

2.3 "while" loop

As well as providing pieces of code which are only performed if a condition is true – conditional code – we may wish to repeat a section of code, time and time-again.

Take, for example, adding up a series of items on a bill. What are the elements needed?

Some statement to say "the following code must be repeated."

Something to say "this is the end of the section to be repeated."

And some way of knowing when the repetition is completed and the program is to continue running beyond the code which has been repeated.

- We use a **while** statement to say "the following code is to be repeated"
- We follow the **while** statement with a block of code in curly braces `{}` – that block indicates how much is to be repeated.
- We place an expression in brackets which yields a boolean result after the word **while**.

Here's an example:

```
public class Loop1
{
    public static void main(String[] args)
    {
        float value_to_date = 0.0F;
        int value_count = 0;
        System.out.println("Adding up a bill");
        System.out.print("Enter cost of first item: ");
        float next_value=WellHouseInput.readNumber();
        while (next_value > 0.0)
        {
            value_to_date = value_to_date + next_value;
            value_count = value_count + 1;
            System.out.print("Enter cost of next item: ");
            next_value=WellHouseInput.readNumber();
        }
        System.out.print("Total value: ");
        System.out.println(value_to_date);
        System.out.print("Item count: ");
        System.out.println(value_count);
    }
}
```

Figure 4 Running public class Loop1

```
bash-2.04$ java Loop1
Adding up a bill
Enter cost of first item290
Enter cost of next item200
Enter cost of next item200
Enter cost of next item0
Total value: 690.0
Item count: 3
bash-2.04$
```

The structure of a **while** loop is just like that of a simple **if** conditional (without the **else**). It's just that the block after the word **if** will be executed 0 or 1 times, and the block after the word **while** will be executed 0 or more times.

As with the **if** statement, the block can be as short or as long as you want, and it can contain other blocks as well.

Every time the execution reaches the end of the block, the condition will be rechecked and the decision made whether to run the block again, or continue execution after the block.

```
public class Loop1
{
    public static void main(String[] args)
    {
        float value_to_date = 0.0F;
        int value_count = 0;
        System.out.println("Adding up a bill");
        System.out.print("Enter cost of first item: ");
        float next_value=WellHouseInput.readNumber();
        while (next_value > 0.0)
        {
```

```

        value_to_date = value_to_date + next_value;
        value_count = value_count + 1;
        System.out.print("Enter cost of next item: ");
        next_value=WellHouseInput.readNumber();
    }
    System.out.print("Total value: ");
    System.out.println(value_to_date);
    System.out.print("Item count: ");
    System.out.println(value_count);
}
}

```

Did you notice that we repeated the prompt for a cost to be entered, and we repeated the code for entering a value?

We can rewrite our code as follows:

```

public class Loop2
{
    public static void main(String[] args)
    {
        float value_to_date = 0.0F;
        int value_count = 0;
        System.out.println("Adding up a bill");
        while (true)
        {
            System.out.print("Enter cost of next item: ");
            float next_value=WellHouseInput.readNumber();
            if (next_value <= 0.0) break;
            value_to_date += next_value;
            value_count++;
        }
        System.out.print("Total value: ");
        System.out.println(value_to_date);
        System.out.print("Item count: ");
        System.out.println(value_count);
    }
}

```

Figure 5 Running public class Loop2

```

bash-2.04$ java Loop2
Adding up a bill
Enter cost of next item490
Enter cost of next item400
Enter cost of next item400
Enter cost of next item400
Enter cost of next item0
Total value: 1690.0
Item count: 4
bash-2.04$

```

What's new?

The boolean expression is always going to continue on around the loop and never jump out:

```
while (true)
```

- In order to get out of the loop, we have instead used a **break** statement:

```
break;
```

which causes us to break out of the innermost loop.

- The **break** statement is within a conditional statement so that we only break out of the loop:
 - From that point in the loop; and
 - Only when the condition is met
- It is common programming practice in loops, such as the one we have seen, to use variables to gather values - accumulators, and/or variables to count. Note the lines:

```
value_to_date = value_to_date + next_value;
value_count = value_count + 1;
```

"Programmers are lazy people," it is said. For something used so frequently, it seems a shame to have to rewrite the variable name twice. Instead, we could write:

```
value_to_date += next_value;
value_count += 1;
```

Other such operators are available as well. For example **-=**, ***=** and **/=**.

We usually count in units of 1, so counters of the type:

```
value_count += 1;
```

would be very common. So common, indeed, that we can rewrite this as:

```
value_count ++ ;
```

or

```
++ value_count ;
```

What's the difference between the two formats?

- When used as a stand-alone statement, nothing.
- When used as part of another statement, if the **++** is in front of the variable name, then the variable is incremented before it is used for anything else in the context. This is called pre-increment.
- And if the **++** follows the variable name, then the variable is incremented after any other use is made. This is called post-increment.

There are also pre- and post-decrements available - taking one off a value using the **--** operator.

2.4 "for" loop

Using a **while** loop for a fixed count is messy. The loop variable is referenced in three separate statements:

```
int count = 1;
while (count < 11)
{
    System.out.println(factorial *= count++);
}
```

which:

- Set the initial value of the loop counter
- Test whether the loop should continue
- Increment the counter

A **for** loop allows us to write a single line which includes all three functions as part of the same statement:

```
for (int count = 1; count<11; count++)
```

Here's a complete example:

```
public class Forloop
{
    public static void main(String[] args)
    {
        int factorial = 1;
        for (int count=1; count < 11; count++)
        {
            System.out.println(factorial *= count);
        }
    }
}
```

Figure 6 Running public class Forloop

```
bash-2.04$ java Forloop
1
2
6
24
120
720
5040
40320
362880
3628800
bash-2.04$
```

The first of the three statements (separated as usual by a semicolon) in the brackets, sets the variable on initial entry to the loop.

The second element is an expression which must yield a boolean result, and is checked on every entry to the loop, after the first or third elements have been performed.

The third element is the action that is performed each time the execution of the code in the loop has been completed, just before the second element test is made to see if the loop must be executed again.

2.5 Labels and breaks

Sometimes a circumstance will occur within a loop and we'll wish to say:

"get me out of this loop"

"go try the loop again"

Of course, we can!

break;

gets you out of a loop

continue;

will go round the loop again.

Let's see an example showing the addition of a series of costs for up to five days. The user enters the cost within a `for` loop, and an inner `while` loop keeps prompting if the values entered are unacceptably high. Once a good value has been entered, the inner loop is broken, or both loops are broken if a zero (used in this example to signify that there is no more data) is entered.

```
public class Bills
{
    public static void main(String[] args)
    {
        float value_to_date = 0.0F;
        int value_count = 0;
        System.out.println("Adding up a bill");
outerloop:
        for (int day=1; day <=6; day++) {
            float next_value = 0.0f;
            while (true) {
                System.out.print("Enter cost for day "+day+": ");
                next_value=WellHouseInput.readNumber();
                if (next_value >= 100.0) {
                    System.out.println("NO WAY - too much");
                    continue;
                }
                if (next_value <= 0.0) break outerloop;
                break;
            }

            value_to_date += next_value;
            value_count++;
        }
        System.out.print("Total value: ");
        System.out.println(value_to_date);
        System.out.print("Item count: ");
        System.out.println(value_count);
    }
}
```

```
bash-2.04$ java Bills
Adding up a bill
Enter cost for day 1 30
Enter cost for day 2 60
Enter cost for day 3 230
NO WAY - too much
Enter cost for day 3 150
NO WAY - too much
Enter cost for day 3 90
Enter cost for day 4 0
Total value: 180.0
Item count: 3
bash-2.04$
```

Figure 7 Running public class Bills

Exercise

One hundred Singapore Dollars are worth 56.54 US Dollars.

Write a program to produce a conversion table. The first column should be a number of US dollars – 10, 20, 30 and so on up to 100. The second column should be the number of Singapore Dollars that it's worth.

You are visiting Singapore and you see a teapot in a shop window at 75 Singapore Dollars. Extend your previous program, adding a third column on the right of your output that says either "YOU CAN AFFORD THE TEAPOT" or "YOU DON'T HAVE ENOUGH MONEY FOR THE TEAPOT"

Take the sample program "Day2a" from the course profile.

- Improve the user messages
- Modify it to accept a maximum of 10 valid inputs
- Modify it so that it does NOT add the terminating "0" into the count

```
public class Day2a {

// =====

public static void main(String [] args) {
short count = 0;
float grand = 0.0f, current = 1.0f;
System.out.println ("A program to calculate a grand total");

while (current != 0.0f) { // Loop to read in values
    current = getvalue();
    grand += current;
    count++;
}
System.out.print("The grand total is ");
System.out.println(grand);
System.out.print("Number of inputs was ");
System.out.println(count);
}

// =====

public static float getvalue() {
    boolean nothingvalid;
    float gotten = 0.0f;
    nothingvalid = true;
    while (nothingvalid) {
        // Loop to keep asking until you get a proper entry!
        System.out.print("Please enter the cost for next day ");
        gotten = WellHouseInput.readNumber();
        if (gotten < 0.0f) {
            System.out.println("Expecetd a +ve no!");
        } else {
            nothingvalid = false;
        }
    }
    return gotten;
}

// =====

}
```

License

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log
Original Version, Well House Consultants, 2004

Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____

License Ends.